An algorithm for automatically obtaining distributed and fault-tolerant static schedules

Alain Girault, Hamoudi Kalla, Mihaela Sighireanu, Yves Sorel

An Algorithm for Automatically Obtaining Distributed and Fault-Tolerant Static Schedules

Alain Girault, Hamoudi Kalla
INRIA, 655 av. de l'Europe
3833 Saint-Ismier, Cedex - France
{Alain.Girault,Hamoudi.Kalla}@inrialpes.fr

Mihaela Sighireanu
LIAFA, Case 7014, 2 place Jussieu
75251 Paris, Cedex 05 - FRANCE
sighirea@liafa.jussieu.fr

Yves Sorel
INRIA, B.P.105
78153 Le Chesnay Cedex - FRANCE
Yves.Sorel@inria.fr

Abstract

Our goal is to automatically obtain a distributed and fault-tolerant embedded system: distributed because the system must run on a distributed architecture; fault-tolerant because the system is critical. Our starting point is a source algorithm, a target distributed architecture, some distribution constraints, some indications on the execution times of the algorithm operations on the processors of the target architecture, some indications on the communication times of the data-dependencies on the communication links of the target architecture, a number $\mathcal{N}pf$ of fail-silent processor failures that the obtained system must tolerate, and finally some real-time constraints that the obtained system must satisfy. In this article, we present a scheduling heuristic which, given all these inputs, produces a fault-tolerant, distributed, and static scheduling of the algorithm on the architecture, with an indication whether or not the real-time constraints are satisfied. The algorithm we propose consist of a list scheduling heuristic based active replication strategy*, that allows at least $\mathcal{N}pf+1$ replicas of an operation to be scheduled on different processors, which are run in parallel to tolerate at most $\mathcal{N}pf$ failures. Due to the strategy used to schedule operations, simulation results show that the proposed heuristic improve the performance of our method, both in the absence and in the presence of failures.*

Keywords: *Fault Tolerance in Distributed and Real-Time Systems, Safety-Critical Systems, software implemented fault-tolerance, multi-component architectures, distribution heuristics.*

1. Introduction

Embedded systems account for a major part of critical applications (space, aeronautics, nuclear…) as well as public domain applications (automotive, consumer electronics…). Their main features are:

- *critical real-time*: timing constraints which are not met may involve a system failure leading to a human, ecological, and/or financial disaster;
- *limited resources*: they rely on limited computing power and memory because of weight, encumbrance, energy consumption (e.g., autonomous vehicles), radiation resistance (e.g., nuclear or space), or price constraints (e.g., consumer electronics);
- *distributed and heterogeneous architecture*: they are often distributed to provide enough computing power and to keep sensors and actuators close to the computing sites.

Moreover, the following aspect, extremely important w.r.t. the target fields, must also be taken into account:

- *fault-tolerance*: an embedded system being intrinsically critical [20], it is essential to insure that its software is fault-tolerant; this in itself can even motivate its distribution; in such a case, at the very least, the loss of one computing site must not lead to the loss of the whole application.

The general domain of our research is that of distributed and fault-tolerant embedded systems. The target applications are critical embedded systems. Our ultimate goal is to produce automatically *distributed and fault-tolerant code* from a given specification of the desired system. In this paper,

we focus on a sub-problem, namely how to produce automatically a *distributed and fault-tolerant static schedule* of a given algorithm on a given distributed architecture.

Concretely, we are given as input a specification of the algorithm to be distributed ($\mathcal{A}lg$), a specification of the target architecture ($\mathcal{A}rc$), some distribution constraints ($\mathcal{D}is$), some information about the execution times of the algorithm blocks on the architecture processors and the communication times of the algorithm data-dependencies on the architecture communication links ($\mathcal{E}xe$), some real-time constraints ($\mathcal{R}tc$), and a number of processor failures ($\mathcal{N}pf$). The goal is to find a static schedule of $\mathcal{A}lg$ on $\mathcal{A}rc$, satisfying $\mathcal{D}is$, and tolerant to at most $\mathcal{N}pf$ processor failures, with an indication whether or not this schedule satisfies $\mathcal{R}tc$ w.r.t. $\mathcal{E}xe$. The global picture is shown in Figure 1. In this paper, we focus on the distribution algorithm.

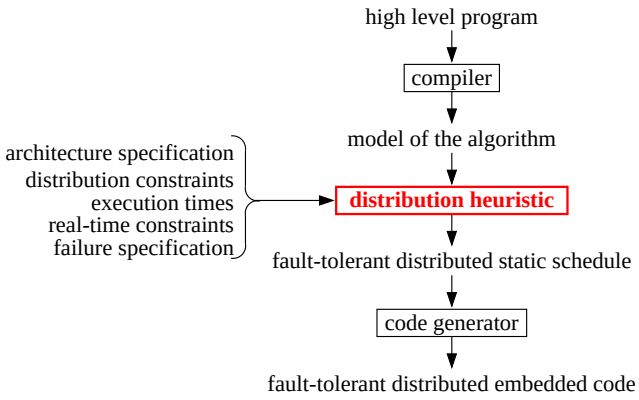

Figure 1. Global picture of our methodology

Finding an algorithm that gives the *best* fault-tolerant schedule w.r.t. the execution times is a well-known NP-hard problem [10]. Instead, we provide a heuristic that gives *one* scheduling, possibly not the best.

There are two constraints we have to deal with:

1. We are targeting *embedded systems*, so first, we *do not* allow the algorithm to add extra hardware, because hardware resources in embedded systems are always limited. It implies that we have to do with the existing parallelism of the given architecture $\mathcal{A}rc$. If the obtained schedule does not satisfy $\mathcal{R}tc$, then it is the responsibility of the user to add more hardware to increase the redundancy. And second, the obtained schedule must be *static* to allow optimisations and to minimise the executive overheads. Therefore, we cannot apply the existing methods, proposed for example in [7, 3, 11], which use preemptive scheduling or approximation methods.
2. We want to obtain the schedule *automatically*, so: The fault-tolerance must be obtained without any help from the user.

For these two reasons, it will fall into the class of *software implemented* fault-tolerance.

2. Related Work

In the literature, we can identify several approaches:

① Some researchers make strong assumptions about the failure models (e.g., only fail-silent) and about the kind of schedule desired (e.g., only static schedule). By adhering to these assumptions however, they are able to obtain automatically distributed fault-tolerant schedules. For instance, Ramamritham requires that the execution cost of each subtask is the same for each processor, and that the communication cost of each data-dependency is the same for each communication link [19], thereby assuming that the target architecture is *homogeneous*. Related approaches can be found in [4] (independent tasks and homogeneous architecture) and [18] (heterogeneous architecture but only one failure is tolerated).

② Other researchers introduce some *dynamicity*. For instance, Caccamo and Buttazzo propose an on-line scheduling algorithm to tolerate task failures on a uniprocessor system [5], while Fohler proposes a mixed on-line and off-line scheduling algorithm to tolerate task failures in a multiprocessor system [9].

③ Finally, some researchers take into account much less restrictive assumptions, but they only achieve *hand-made* solutions, e.g., with specific communication protocols, voting mechanisms... See the vast literature on general fault-tolerance, for instance [17].

Like the other researchers belonging to the first group, we propose an automatic solution to the fault-tolerance distributed problem. The conjunction of the four following points makes our approach original:

1. We take into account the execution time of both the computation operations and the data communications to optimise the critical path of the obtained schedule.
2. Since we produce a static schedule, we are able to compute the expected completion date for any given operation or data communication, both in the presence and in the absence of failures. Therefore we are able to check the real-time constraints $\mathcal{R}tc$ before the execution. If $\mathcal{R}tc$ is not satisfied, we can give a warning to the designer, so that he can decide whether to add more hardware or to relax $\mathcal{R}tc$.
3. The given algorithm $\mathcal{A}lg$ can be designed with a high-level programming language based on a formal mathematical semantics. This is for instance the case of synchronous languages, which are moreover well suited to the programming of embedded critical systems [15, 2]. The advantage is that $\mathcal{A}lg$ can be formally verified with model-checking and theorem proving tools, and

therefore we can assume safely that it is free of design faults. The scheduling method we propose in this paper preserves this property.

4. Operations scheduled on the distributed architecture are guaranteed to complete if at most $\mathcal{N}pf$ processors fails at any instant of time. There is no need for a complex failure detection mechanism, and in particular we do not need timeouts to detect the processor failures; there is no need for the processors to propagate the state of the faulty ones; and finally, due to the scheduling strategy used the time needed for handling a failure is minimal.

A *different* version of the method presented here has been published as an *abstract* in [12] and as a full version in a *workshop* [13]. It is different since it addresses distributed architectures consisting of several nodes connected to a single *bus*, while here we address more general distributed architectures since they can include point-to-point communication links (see Section 3.3). As a result, here the communications can be scheduled in parallel on the communication links, and the fault-tolerance is achieved with the *software redundancy* of both the computation operations and the data communications (see Section 4.1). In [12, 13] we used the *time redundancy* of the data communications. Also, we can cope with intermittent processor failures and we do not need to use timeouts to detect failures, which was not the case in [12, 13]. In conclusion, the method presented here is *complementary* and *more general* than the one presented in [12, 13].

There is another work involving some of the authors [8], where a totally different approach is taken: First, communication link failures are also taken into account, and second, the method presented involves building a basic schedule for each possible failure, and then merging these basic schedules to obtain a distributed fault-tolerant schedule. The method presented here is lighter, faster, and more efficient, but it only copes with processor failures.

The rest of the paper is organised as follows. Section 3 states our fault-tolerance problem, and presents the various models used by our method. Section 4 presents the proposed solution for providing fault-tolerance. Section 5 provide a correctness proof of the proposed algorithm. Simulation results are presented in Section 6. Finally, Section 7 concludes and proposes directions for future research.

3. Models

3.1. Failure Model

As said in the introduction, our goal is to find a static schedule of $\mathcal{A}lg$ on $\mathcal{A}rc$, satisfying $\mathcal{D}is$, and tolerant to at most $\mathcal{N}pf$ processor failures, with an indication whether or not this schedule satisfies $\mathcal{R}tc$ w.r.t. $\mathcal{E}xe$. The failures considered are fail-silent processor failures (permanent as well as intermittent). By "tolerant" we mean that the obtained schedule must achieve "failure masking" [17]. More precisely, this will be done by means of error compensation, using software redundancy. The real-time constraints $\mathcal{R}tc$ can be, for instance, a deadline for the completion date of the whole schedule. If the user wants to be more precise, he/she can specify a deadline on the completion date of a particular sub-task of the algorithm. The fact that the obtained schedule is static allows the computation of its completion date w.r.t. $\mathcal{E}xe$.

3.2. Algorithm Model

The algorithm is modelled by a *data-flow graph*. Each vertex is an *operation* and each edge is a *data-dependency*. The algorithm is executed repeatedly for each input event from the sensors in order to compute the output events for actuators. We call each execution of the data-flow graph an *iteration*. This cyclic model exhibits the potential parallelism of the algorithm through the partial order associated to the graph. This model is commonly used for embedded systems and automatic control systems.

Operations of the graph can be either:

- a computation operation (`comp`): its inputs must precede its outputs; the outputs depend only on the input values; there is no internal state variable and no other side effect;
- a memory operation (`mem`): the data is held by a `mem` in sequential order between iterations; the output precedes the input, like a register in Boolean circuits;
- an external input/output operation (`extio`). Operations with no predecessor in the data flow graph (resp. no successor) are the external input interfaces (resp. output), handling the events produced by the sensors (resp. actuators). The `extios` are the only operations with side effects; however, we assume that two executions of a given input `extio` in the same iteration always produce the same output value.

Figure 2 is an example of algorithm graph, with nine operations: I and O are `extios` (resp. input and output), while A–G are `comps`. The data-dependencies between operations are depicted by arrows. For instance the data-dependency A $\triangleright$ B can correspond to the sending of some arithmetic result computed by A and needed by B.

3.3. Architecture Model

The architecture is modelled by a graph, where each vertex is a processor, and each edge is a communication link.

Classically, a processor is made of one computation unit, one local memory, and one or more communication units, each connected to one communication link. Communication units execute data transfers, called `comms`. The chosen communication mechanism is the send/receive [14], where the send operation is non-blocking and the receive operation blocks in the absence of data. Figure 2 is an example of architecture graph, with three processors and three point-to-point links.

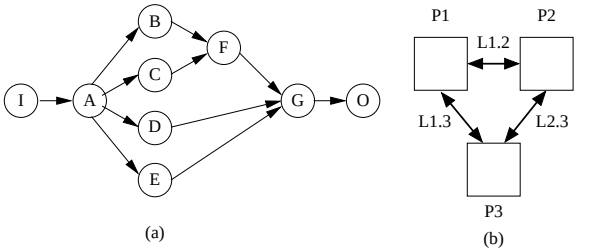

Figure 2. Example of (a) an algorithm graph $\mathcal{A}lg$ (a); and (b) an architecture graph $\mathcal{A}rc$

3.4. Distribution Constraints, Execution Times, and Real-Time Constraints

For the operations, the execution times $\mathcal{E}xe$ consist of a table associating to each pair $\langle o, p\rangle$ the execution time of the operation o on the processor p, expressed in time units. Since the target architecture is heterogeneous, the execution times for a given operation can be distinct on each processor. Specifying the distribution constraints $\mathcal{D}is$ involves associating the value "∞" to certain pairs $\langle o, p\rangle$, meaning that o cannot be executed on p.

For the inter-processor communications, the execution times $\mathcal{E}xe$ consist of a table associating to each pair $\langle data\ dependency, communication\ link\rangle$ the value of the transmission time of this $data\ dependency$ on this $communication\ link$, again expressed in time units.

time	operation								
proc.	I	A	B	C	D	E	F	G	O
P1	1	2	3	2	3	1	2	1.4	1.4
P2	1.3	1.5	1	3	1.7	1.2	2.5	1	∞
P3	∞	1	1.5	1	3	2	1	1.5	1.8

Table 1. Distributed constraints $\mathcal{D}is$ and execution times $\mathcal{E}xe$ for operations

For instance, the $\mathcal{D}is$ and $\mathcal{E}xe$ for $\mathcal{A}lg$ and $\mathcal{A}rc$ of Figure 2 are given by the two tables 1 and 2. Here it takes more time to communicate the data-dependency I $\triangleright$ A than A $\triangleright$ B simply because there are more data to transmit. The point-to-point links $\{$L1.2$\}$ and $\{$L1.3, L2.3$\}$ are heterogeneous. This table only gives the transmission times for *inter-processor* communications. For an *intra-processor* communication, the time is always 0.

time	data-dependency					
link	I $\triangleright$ A	A $\triangleright$ B	A $\triangleright$ C	A $\triangleright$ D	A $\triangleright$ E	B $\triangleright$ F
L1.2	1.75	1	1	1.5	1	1
L2.3	1.25	0.5	0.5	1	0.5	0.5
L1.3	1.25	0.5	0.5	1	0.5	0.5

time	data-dependency				
link	C $\triangleright$ F	D $\triangleright$ G	E $\triangleright$ G	F $\triangleright$ G	G $\triangleright$ O
L1.2	1.3	1.9	1.3	1	1.1
L2.3	0.8	1.4	0.8	0.5	0.6
L1.3	0.8	1.4	0.8	0.5	0.6

Table 2. Execution times $\mathcal{E}xe$ for communications

Finally, the real-time constraints $\mathcal{R}tc$ are also given in time units. They can be, for instance, a deadline for the completion date of the whole schedule. For our example, we will take $\mathcal{R}tc = 16$, which means that the obtained static fault-tolerant distributed schedule must complete in less than 16 time units.

4. The Proposed Solution

In this Section we discuss some of the basic principles used in the proposed approach, followed by a description of our algorithm. The algorithm we propose is a list scheduling heuristic based *active replication strategy* [6], that allows at least $\mathcal{N}pf$+1 replicas of an operation to be scheduled on different processors, which are run in parallel to tolerate at most $\mathcal{N}pf$ processors failures.

4.1. Algorithm Principle

The proposed solution uses the software redundancy of both `comps`/`mems`/`extios` and of `comms`. Each operation X of the algorithm graph is replicated on $\mathcal{R}ep$ different processors of the architecture graph, where $\mathcal{R}ep \geq \mathcal{N}pf + 1$. Each of these $\mathcal{R}ep$ replicas send their results in parallel to all the replicas of all the successor operations in the data-flow graph. Therefore, each operation will receive its set of inputs $\mathcal{R}ep$ times; as soon as it receives the first set, the operation is executed and ignores the later inputs. However, in some cases, the replica of an operation will only receive some of its inputs *once*, through an intra-processor communication. For the sake of simplicity, suppose we have an operation X with only one input produced by its predecessor Y (see Figure 3(a)).

Consider the replica of X which is assigned to processor P. Two cases can arise: either one replica of Y is also scheduled on P, or all the replicas of Y are assigned to processors distinct from P. In the first case, the `comm` from Y to

X will not be replicated and will be implemented as a single *intra-processor* communication (see Figure 3(b)). Indeed, the replicas of this comm would only be used if P failed, but in this case the replica of X assigned to P would not need this input. In the second case, the comm from Y to X will be replicated $\mathcal{N}pf$+1 times, each implemented as an *inter-processor* communication (see Figure 3(c)).

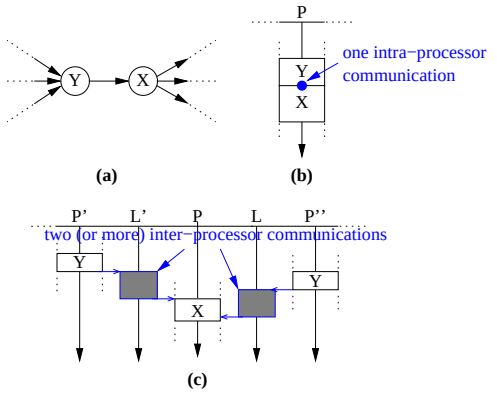

Figure 3. (a) Algorithm sub-graph; (b) At least one replica of Y is on P; (c) No replica of Y is on P.

Figure 3 illustrates this example by showing the partial schedules obtained for the X and Y subgraph. In these diagrams, an operation is represented by a white box, whose height is proportional to its execution time. A comm is represented by a gray box, whose height is proportional to its communication time, and whose ends are bound by two arrows: one from the source operation and one to the destination operation.

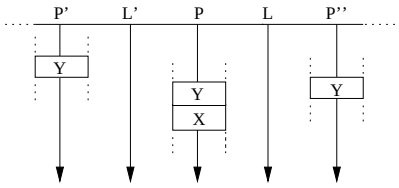

Figure 4. Schedule more than $\mathcal{N}pf$ replicas of an operation.

Since the communication cost between operations assigned to the same processor is considered to be negligible, replicating an operation more than $\mathcal{N}pf$ +1 times reduces the global interprocessor communication overheads of the schedule. Consider the schedule of Figure 3(c): if Y is replicated on P, the schedule length can be reduced, both in the presence and in the absence of failures, as shown in Figure 4.

4.2. Scheduling Heuristic

The heuristic implementing this solution is a greedy list scheduling [22], called the Fault-Tolerance Based Active Replication strategy (FTBAR) algorithm. We present the scheduling algorithm in macrosteps, the superscript number in parentheses refers to the step of the heuristic, e.g., $O^{(n)}_{sched}$.

Before describing the heuristic, we define the following notations which are used in the rest of this paper:

- $O^{(n)}_{cand}$: The list of *candidate* operations, this list is built from the algorithm graph vertices. An operation is said to be a candidate if all its predecessors are already scheduled.
- $O^{(0)}_{sched}$: The list of *scheduled* operations.
- $pred(o_i)$: The set of predecessors of operation o_i.
- $succ(o_i)$: The set of successors of operation o_i.
- $R^{(n)}$: The critical path length.
- $E^{(n)}_{exc}(o_i, p_j)$: The end execution time of operation o_i scheduled on processor p_j.
- $E^{(n)}_{com}(o_i, o_j)$: The end of data communication time from operation o_i to operation o_j.
- $\overline{S}^{(n)}(o_i)$ is the latest start time from end of o_i.
- $S^{(n)}_{best}(o_i, p_l)$: The earliest time at which operation o_i can start execution on processor p_l. It is computed as follows:
$$S^{(n)}_{best}(o_i, p_l) = \max_{o_j \in pred(o_i)} \left\{ \min_{k=1}^{Npf+1} E^{(n)}_{com}(o^k_j, o_i) \right\}$$
where o^k_j is the k^{th} replica of o_j.
If o_i and o_j are scheduled in the same processor p_l then $E^{(n)}_{com}(o^k_j, o_i) = E^{(n)}_{exc}(o_j, p_l)$.
- $S^{(n)}_{worst}(o_i, p_l)$: The earliest time at which operation o_i can start execution on processor p_l, taking into account all the predecessors replicas. It is computed as follows:
$$S^{(n)}_{worst}(o_i, p_l) = \max_{o_j \in pred(o_i)} \left\{ \max_{k=1}^{Npf+1} E^{(n)}_{com}(o^k_j, o_i) \right\}$$
where o^k_j is the k^{th} replica of o_j.
If o_i and o_j are scheduled in the same processor p_l then $E^{(n)}_{com}(o^k_j, o_i) = E^{(n)}_{exc}(o_j, p_l)$.

The *schedule pressure* [21] is used as a cost function to select the best operation/processor pair. The schedule pressure noted by $\sigma^{(n)}(o_i, p_j)$ tries to minimise the length of the critical path of the algorithm and to exploit the scheduling margin of each operation. It is computed for each processor $p_j \in P$ (P is the processor's set) and each operation $o_i \in O^{(n)}_{cand}$ by using two functions:

1. The *schedule-flexibility* SF is defined as:

$$SF^{(n)}(o_i, p_j) = R^{(n)} - S^{(n)}_{worst}(o_i, p_j) - \overline{S}^{(n)}(o_i)$$

2. The *schedule-penalty* SP is defined as:

$$SP^{(n)}(o_i, p_j) = R^{(n)} - R^{(n-1)}$$

With these two functions, the schedule pressure σ is computed as follows:

$$\begin{aligned}\sigma^{(n)}(o_i, p_j) = &\quad SP^{(n)}(o_i, p_j) - SF^{(n)}(o_i, p_j) \\ = &\quad S^{(n)}_{worst}(o_i, p_j) + \overline{S}^{(n)}(o_i) - R^{(n-1)}\end{aligned}$$

The schedule pressure measures how much the scheduling of the operation lengthens the critical path of the algorithm. Therefore it introduces a priority between the operations to be scheduled. Note that, since all candidates operations at step n have the same value $R^{(n-1)}$, it is not necessary to compute $R^{(n-1)}$.

The FTBAR fault-tolerance scheduling heuristic is formally described below:

The FTBAR Algorithm:
begin
Initialise the lists of candidate and scheduled operations:
$O^{(0)}_{cand} := \{o \in O \mid pred(o) = \emptyset\}$;
$O^{(0)}_{sched} := \emptyset$;
while $O^{(n)}_{cand} \neq \emptyset$ **do**

① Compute the schedule pressure for each operation o_i of $O^{(n)}_{cand}$ on each processor p_j using $S^{(n)}_{worst}$, and keep the first $\mathcal{N}pf$+1 min results for each operation:
$\forall o_i \in O^{(n)}_{cand}$,
$\cup_{l=1}^{l=\mathcal{N}pf+1} \sigma^{(n)}_{best}(o_i, p_{i_l}) := \min_{p_j \in P}^{\mathcal{N}pf+1} \sigma^{(n)}(o_i, p_j)$;

② Select the best candidate operation o such that:
$\sigma^{(n)}_{urgent}(o) := \max_{o_i \in O^{(n)}_{impl}} \cup_{l=1}^{l=\mathcal{N}pf+1} \sigma^{(n)}_{best}(o_i, p_{i_l})$;

③ Apply $Minimize\_start\_time$ for the best candidate operation o on the first $\mathcal{N}pf$+1 processors computed at ①;

④ Update the lists of candidate and scheduled operations:
$O^{(n)}_{sched} := O^{(n-1)}_{sched} \cup \{o\}$;
$O^{(n+1)}_{cand} := O^{(n)}_{cand} - \{o\} \cup \overline{Succ}\{o\}$; with:
$\overline{Succ}\{o\} = \{o' \in succ(o) \mid pred(o') \subseteq O^{(n)}_{sched}\}$;

end while
end

Initially, $O^{(0)}_{sched}$ is empty and $O^{(0)}_{cand}$ is the list of operations without any predecessors. At the n-th step ($n \geq 1$), the list of *already scheduled* operations $O^{(n)}_{sched}$ is kept. Also, the list of *candidate* operations $O^{(n)}_{cand}$ is built from the algorithm graph vertices.

At each step n, one operation of the list $O^{(n)}_{cand}$ is selected to be scheduled. To select an operation, we select at the micro-step ①, for each operation o_i, the $\mathcal{N}pf+1$ processors having the minimum schedule pressure. Then among those best pairs $\langle o_i, p_j \rangle$, we select at the micro-step ② the one having the maximum schedule pressure, i.e., the most urgent pair.

The selected operation is implemented at the micro-step ③ on the $\mathcal{N}pf$+1 processors computed at micro-step ①, and the `comms` implied by this implementation are also implemented. At this micro-step the start time of the selected operation o is reduced by replicating its predecessors using a procedure *Minimise_start_time* proposed by Ahmad and al. in [1], which is formally described below:

Minimise_start_time(o,p):
begin

❶ Determine earliest start time $S^{(n)}_{worst}(o, p)$;

❷ **if** $S^{(n)}_{worst}(o, p)$ is undefined **then** quit because o cannot be scheduled on p;

❸ Find out the Latest Immediate Predecessor (LIP) of o;

❹ Minimize the start time of this LIP by recursively calling *Minimize_start_time(LIP,p)*;

❺ Compute the $new\ S^{(n)}_{worst}(o, p)$;

❻ **if** ($new\ S^{(n)}_{worst}(o, p) \geq S^{(n)}_{worst}(o, p)$)

❼ **then**

- Undo all the replications just performed in ❹;
- Schedule o to p at $S^{(n)}_{best}(o, p)$;
- The `comms` implied by (b) are also implemented here such that, each replica of o receives data from each replica of these predecessors o_j through parallel links;

❽ **else** Find out the new LIP of o and repeat from ❹;

end

For each pair $\langle$predecessor, operation replica$\rangle$, `comms` are added in parallel links if and only if all the replicas of the predecessor are on different processors. If this is not the case, i.e., if there exists a replica of the predecessor on the same processor, no `comm` is added (see Section 4.1 and Figure 3).

When a `comm` is generated, it is assigned to the set of communication units bound to the communication medium connecting the processors executing the source and destination operations. At the end, all the `comms` assigned to the same communication unit are statically scheduled. The `comms` are thus totally ordered over each communication medium. Provided that the network preserves the integrity and the ordering of messages, this total order of the `comms` guarantees that data will be transmitted correctly between processors. The obtained schedule also guarantees a deadlock free execution.

The strategy used to schedule operations ensures a minimum run-time overhead in the faulty system (a system presenting at least one failure) by using $S^{(n)}_{worst}(o, p)$ to give priority to operations and $S^{(n)}_{best}(o, p)$ to schedule operations.

4.3. An Example

We have implemented our fault-tolerant heuristic in the SYNDEX [21] tool, which is a tool for optimizing the implementation of real-time embedded applications on multicomponent architecture.

We apply our heuristic to the example of Figure 2. The user requires the system to tolerate one permanent processor failure, i.e., $\mathcal{N}pf = 1$. The execution characteristics of each `comp`/`mem`/`extio` and `comm` are specified by the two tables of time units given in Section 3.4.

After the first two steps of our heuristic, we obtain the temporary schedule of Figure 5.

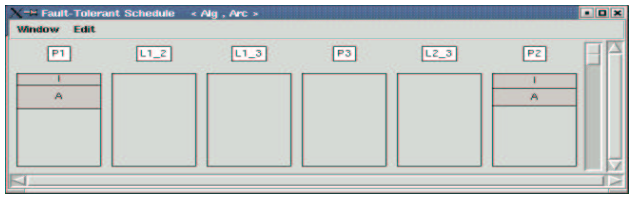

Figure 5. Step 2

In the next step, operation C is scheduled. Assigning C to $P1$, $P2$ and $P3$ gives an expected *schedule pressure* of 9.73, 10.53 and 9.23 respectively. But, if A, the LIP of C, is duplicated to $P3$, the schedule pressure of C can be reduced to 5.73, which means that the start time of C is also reduced. We therefore schedule a new replica of A on $P3$ and two replicas of C on $P3$ and $P1$, which minimizes the schedule pressure. As shown in Figure 6, operation A receives its inputs data twice from the replicas of I scheduled on $P1$ and $P2$, and the start time of A is the end of the earliest communication between $\langle I, A\rangle$ on $\{\text{L1\_3}\}$ and $\{\text{L2\_3}\}$. We obtain therefore the temporary schedule of Figure 6.

Figure 6. Step 3

Similarly, operations B, D, E, F, G, and O are scheduled. At the end of our heuristic, we obtain the final schedule presented in Figure 7. Each operation of the algorithm graph is replicated at least twice and these replicas are assigned to different processors. More important, the real-time constraint is satisfied since the total time is $15.05 < \mathcal{R}tc$.

Figure 7 shows that some communications are not useful in the absence of failures. For example, the communication of the result of the operation $\langle \text{I}, \text{P2}\rangle$ to the operation $\langle \text{A}, \text{P3}\rangle$ is not used since the result sent by $\langle \text{I}, \text{P1}\rangle$ arrives first. However, these communications may become useful during a faulty execution. For instance, suppose that P1 crashes at time0 (see Figure 8). Since we have assumed a fail-silent model, P1 fails to produce its expected results, the output of $\langle \text{I}, \text{P1}\rangle$ that should have been sent to $\langle \text{A}, \text{P3}\rangle$ and the output of $\langle \text{C}, \text{P1}\rangle$ that should have been sent to $\langle \text{F}, \text{P2}\rangle$. Therefore the failure of P1 can actually be detected by P3 only after the expected completion date of the `comm` from $\langle \text{I}, \text{P1}\rangle$ to $\langle \text{A}, \text{P2}\rangle$. Detecting P1's failure is useful in order to avoid sending further `comms` to P1, but functionally, we do not need it: indeed, the static schedule *transparently* tolerates one failure since $\mathcal{N}pf = 1$. Actually, it is not entirely transparent since the resulting schedule has a greater execution time.

Figure 7. Final fault-tolerant schedule

Figure 8 shows the schedule when P1 crashes. As expected, the data sent by all the `comms` toward the faulty processor P1 are discarded. The schedule corresponding to the subsequent iterations is the same except that all the operations scheduled on P1 as well as all the `comms` from and to P1 have disappeared. Finally, the real-time constraint is still satisfied since the total time is respectively 15.35, 15.05, 12.6 when P1, P2, or P3 fails at time 0.

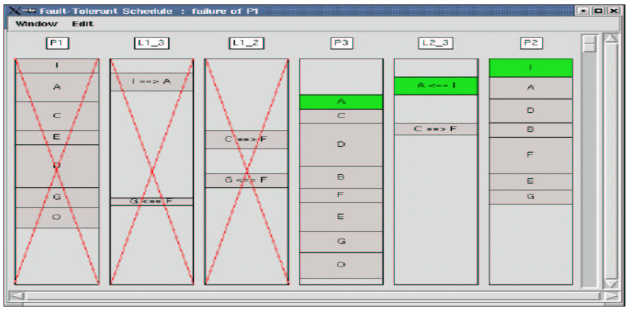

Figure 8. Timed execution when P1 crashes

4.4. Analysis of the Example

To evaluate the overheads introduced by the fault-tolerance, let us consider the non fault-tolerant schedule produced for our example with a basic scheduling heuristic (for instance the one of SYNDEX). The schedule length generated by this heuristic is 10.7.

Neither this non fault-tolerant schedule nor the fault-tolerant schedule of Figure 7 are the best possible. Remember that finding the best schedule is an NP-hard problem; this is the reason why we have designed a heuristic scheduling algorithm. In this particular case, the fault-tolerance overheads is therefore $15.05 - 10.7 = 4.35$.

In the fault-tolerant schedule, some communications take place although they are not necessary. On the other hand, the response time of the faulty system is minimized, since results are sent without waiting for any timeout (see Figure 8). For the same reason, the system supports the arrival of several failures during the same iteration since there are no risks that the sum of pending timeouts overtakes the desired real-time constraints. In other words, we do not need to make any assumptions on the failure inter-arrival time.

This solution is appropriate to an architecture where the communication means are *point-to-point* links, which allow parallel communications to take place. For multi-point links, the overheads introduced by the replication of `comms` may be too high because of their serialization on a single link.

5. Runtime Behavior

In our heuristic, $\mathcal{N}pf$ faults can be tolerated by scheduling $\mathcal{N}pf$+1 replicas for each operation on different processors. We assume in this paper that all values returned by the $\mathcal{N}pf$+1 replicas of any given input operation are identical in the same iteration. If no fault occurs, each of the $\mathcal{N}pf$+1 replicas of an operation receives its inputs in parallel from all the replicas of its predecessor operations in the data-flow graph; as soon as it receives the first set, the operation is executed and ignores the later $\mathcal{N}pf$ inputs. If there are k permanent faults ($k \leq \mathcal{N}pf$), each replica of an operation scheduled on a non-faulty processor receives its inputs in parallel from all the replicas of its predecessors operations scheduled on a non-faulty processors; as soon as it receives the first set, the operation is executed and ignores the later inputs. Concerning the failure detection, there are two options:

1. Either we do not perform any failure detection, in which case, after a failure, the remaining processors will continue to send results to the faulty one. This will not help reducing the communication overheads, especially when the `comms` have to be serialized over a multi-point communication link. The advantage is that if a processor experiences an *intermittent* failure, then since it will continue to receive inputs from the healthy processors, it will be able to produce its results again when recovering from its intermittent failure.
2. Or we perform a failure detection procedure by knowing at what time each `comm` is supposed to happen, and by deciding accordingly that when a `comm` did not happen,then the sending processor is faulty. Each processor can therefore maintain an array of faulty processors and avoid further `comms` to the faulty processors in both the remaining of the transient iteration and the subsequent iterations. The drawback is that an intermittent failure cannot be recovered. Indeed, when a processor is detected to be faulty, the other healthy processors will update their array of faulty processors, and will not send any more data during the subsequent iterations. So even if this faulty processor comes back to life, it will not receives any inputs and will not be able to perform any computation. Therefore, in the subsequent iterations, it will fail to send any data on its communication links, and the other healthy processors will never be able to detect that it came back to life. The same applies to failure detection mistakes.

The choice between these two options can be left to the user. It will depend on the intermittent failure rate of the application as well as on the topology and the bandwidth of the network.

6. Performance Evaluation

To evaluate our faut-tolerant scheduling heuristic, we have compared the performance of the proposed algorithm with the algorithm proposed by Hashimoto and al. in [16], called HBP (Height-Based Partitioning) which is the closest to FTBAR that we have found in the literature. Since, HBP assumes homogeneous systems and only use software redundancy of the algorithm's operations, FTBAR is downgraded to these assumptions to make the comparison meaningful. The goal of our simulations is to compare the fault-tolerance overheads of HBP and FTBAR, both in the absence and in the presence of one processor failure.

6.1. Simulation Parameters

We have applied FTBAR and HBP heuristics to a set of random algorithm graphs with a wide range of parameters. A random algorithm graph is generated as follows: Given the number of operations N, we randomly generate a set of levels with a random number of operations. Then, operations at a given level are randomly connected to operations at a higher level. The execution times of each operation are randomly selected from a uniform distribution with the mean equal to the chosen average execution time. Similarly, the communication times of each data dependency are randomly selected from a uniform distribution with the mean equal to the chosen average communication time.

For generating the complete set of algorithm graphs, we vary two parameters: $N = 10, 20, ..., 80$, and the

communication-to-computation ratio, defined as the average communication time divided by the average computation time, $CCR = 0.1, 0.5, 1, 2, 5, 10$.

6.2. Performance Results and Analysis

We present in this Section the performance results on the fault-tolerance heuristic. We compute the fault-tolerance overhead in the following way:

$$Overheads = \frac{(FTSL) - (non\ FTSL)}{(FTSL)} \times 100$$

where the *non FTSL (Fault Tolerant Schedule Length)* is produced by FTBAR with $\mathcal{N}pf = 0$.

We have plotted in Figures 9 and 10 the average fault-tolerance overheads (averaged over 60 random graphs) as a function of N and CCR, both in the absence (Figure 9(a) and 10(a)) and in the presence of one processor failure (Figure 9(b) and 10(b); here we have computed the average overheads when each of the four processors fails, and plotted the max overheads over these four processors).

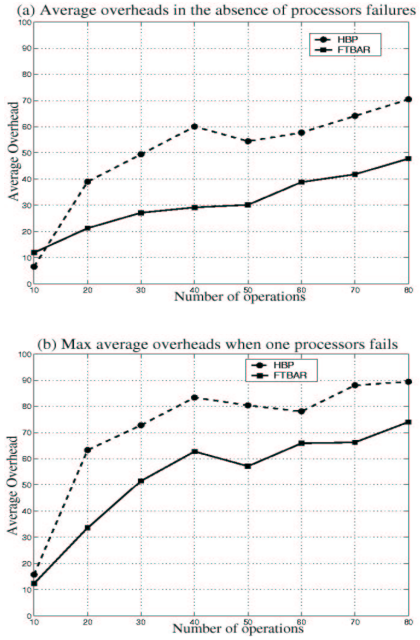

Figure 9. Impact of the number of operations for $\mathcal{N}pf = 1$, $P = 4$ and $CCR = 5$

Figure 9 shows that average overheads increases with N. This is due to the active replication of all operations and communications. Figure 9 also shows that FTBAR perform better than HBP.

Figure 10 shows that, when the average communication time is strictly greater than the average execution time, the average overheads decrease. For $CCR \leq 1$, there is little difference between HBP and FTBAR. In contrast, for $CCR \geq 2$, FTBAR performs significantly better than HBP (by at least 20%). This is due to our *schedule pressure* which tries to minimize the length of the critical path.

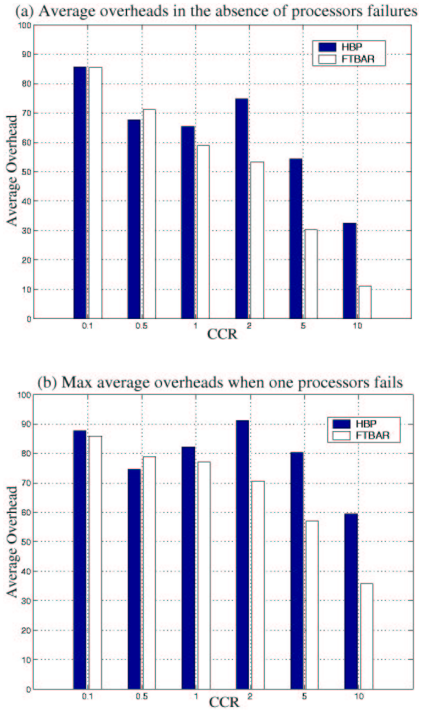

Figure 10. Impact of the communication-to-computation ratio for $\mathcal{N}pf = 1$, $P = 4$ and $N = 50$

The time complexity of FTBAR is less than the time complexity of HBP. The reason is that HBP investigates more possibilities than FTBAR when selecting the processor for a candidate operation.

7. Conclusion and Future Work

The literature about fault-tolerance of distributed and/or embedded real-time systems is very abundant. Yet, there are few attempts to combine fault-tolerance and automatic generation of distributed code for embedded systems.

In this paper, we have studied this problem and proposed a software implemented fault-tolerance solution.

We have proposed a new scheduling heuristic, called FTBAR (Fault-Tolerance Based Active Replication), that produces automatically a static distributed fault-tolerant schedule of a given algorithm on a given distributed architecture. Our solution is based on the software redundancy of both the computation operations and the communications. All replicated operations send their results but only the one which is received first by the destination processor is used; the other results are discarded. The implementation uses a

scheduling heuristic for optimizing the critical path of the distributed algorithm obtained. It is best suited to architectures with point-to-point links. There are some communication overheads, but on the other hand, several failures in a row can be tolerated. Also, depending on the failure detection mechanism chosen, intermittent failures can be tolerated as well.

We have implemented our FTBAR heuristic within the SYNDEX tool. SYNDEX is able to generate automatically executable distributed code, by first producing a static distributed schedule of a given algorithm on a given distributed architecture, and then by generating a real-time distributed executive implementing this schedule. We have also implemented the HBP (Height-Based Partitioning [16]) heuristic. Although HBP only considers homogeneous architectures and only tolerates one processor failure, it is the closest to our work that we have found in the literature. The experimental results shows that FTBAR performs better than HBP, both in the absence and in the presence of failures.

Currently, we are performing extensive benchmark testing of FTBAR on heterogeneous architectures. The first results show that the overheads increases with the number of failures $\mathcal{N}pf$.

Finally, our solution can only tolerate processor failures. We are currently working on new solutions to take communication link failures and reliability into account. We also plan to experiment our method on an electric autonomous vehicle, with a 5-processor distributed architecture.

Acknowledgments

The authors would like to thank Cătălin Dima, Thierry Grandpierre, Claudio Pinello, and David Powell for their helpful suggestions.

References

[1] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. In *IEEE Transactions on Parallel and Distributed Systems*, volume 9, pages 872–892, September 1998.

[2] G. Berry and A. Benveniste. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[3] A. Bertossi and L. Mancini. Scheduling algorithms for fault-tolerance in hard-real-time systems. *Real-Time Systems Journal*, 7(3):229–245, 1994.

[4] A. Bertossi, L. Mancini, and F. Rossini. Fault-tolerant rate-monotonic first-fit scheduling in hard real-time systems. *IEEE Trans. on Parallel and Distributed Systems*, 10:934–945, 1999.

[5] M. Caccamo and B. Buttazzo. Optimal scheduling for fault-tolerant and firm real-time systems. In *5th International Conference on Real-Time Computing Systems and Applications*. IEEE, Oct. 1998.

[6] P. Chevochot and I. Puaut. Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategie. In *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 356–363, HongKong, China, December 1999.

[7] J.-Y. Chung, J. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Trans. on Computers*, 39(9):1156–1174, September 1990.

[8] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel. Off-line real-time fault-tolerant scheduling. In *9th Euromicro Workshop on Parallel and Distributed Processing, PDP'01*, pages 410–417, Mantova, Italy, February 2001.

[9] G. Fohler. Adaptive fault-tolerance with statically scheduled real-time systems. In *Euromicro Workshop on Real-Time Systems, EWRTS'97*, Toledo, Spain, June 1997. IEEE.

[10] M. Garey and D. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman Company, San Francisco, 1979.

[11] S. Ghosh. *Guaranteeing Fault-Tolerance through Scheduling in Real-Time Systems*. PhD Thesis, University of Pittsburgh, 1996.

[12] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. Fault-tolerant static scheduling for real-time distributed embedded systems. In *21st International Conference on Distributed Computing Systems, ICDCS'01*, pages 695–698, Phœnix, USA, April 2001. IEEE. Extended abstract.

[13] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. Generation of fault-tolerant static scheduling for real-time distributed embedded systems with multi-point links. In *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, FTPDS'01*, San Francisco, USA, April 2001. IEEE.

[14] M. Gupta and E. Schonberg. Static analysis to reduce synchronization cost in data-parallel programs. In *23rd Symposium on Principles of Programming Languages*, pages 322–332, January 1996.

[15] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.

[16] K. Hashimoto, T. Tsuchiya, and T. Kikuno. Effective scheduling of duplicated tasks for fault-tolerance in multiprocessor systems. *IEICE Transactions on Information and Systems*, E85-D(3):525–534, March 2002.

[17] P. Jalote. *Fault-Tolerance in Distributed Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[18] X. Qin, Z. Han, H. Jin, L. P. Pang, and S. L. Li. Real-time fault-tolerant scheduling in heterogeneous distributed systems. In *Proceeding of the International Workshop on Cluster Computing-Technologies, Environments, and Applications (CC-TEA'2000)*, Las Vegas, USA, June 2000.

[19] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Trans. on Parallel and Distributed Systems*, 6(4):412–420, April 1995.

[20] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and Systems Safety*, 43(2):189–219, 1994. Research Report CSL-93-01.

[21] A. Vicard. *Formalisation et Optimisation des Systmes Informatiques Distribus Temps-Rel Embarqus*. PhD Thesis, University of Paris XIII, July 1999.

[22] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.